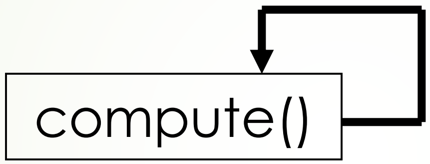
Recursion

Introduction to Recursion

Introduction to Recursion

- Recursive Method:
  - A recursive method is a method that calls itself.



Introduction to Recursion

- Recursion can be used to manage repetition.

- Recursion is a process in which a module achieves a repetition of algorithmic steps by calling itself.

- Each recursive call is based on a different, generally simpler, instance.

Introduction to Recursion

- Recursion is something of a divide and conquer, top-down approach to problem solving.

  - It divides the problem into pieces or selects out one key step, postponing the rest.

4 Fundamental Rules :

1. Base Case : Always have at least one case that can be solved without recursion.
2. Make Progress : Any recursive call must progress towards a base case.
3. Always Believe : Always assume the recursive call works.
4. Compound Interest Rule : Never duplicate work by solving the same instance of a problem in separate recursive calls.

Basic Form :

```
void recurse ()
{
   recurse ();              //Function calls itself
}

int main ()
{
   recurse ();             //Sets off the recursion
}
```

How does it work?

1. The module calls itself.
2. New variables and parameters are allocated storage on the stack.
3. Function code is executed with the new variables from its beginning. It does not make a new copy of the function. Only the arguments and local variables are new.
4. As each call returns, old local variables and parameters are removed from the stack.
5. Then execution resumes at the point of the recursive call inside the function.

Why use Recursive Methods?

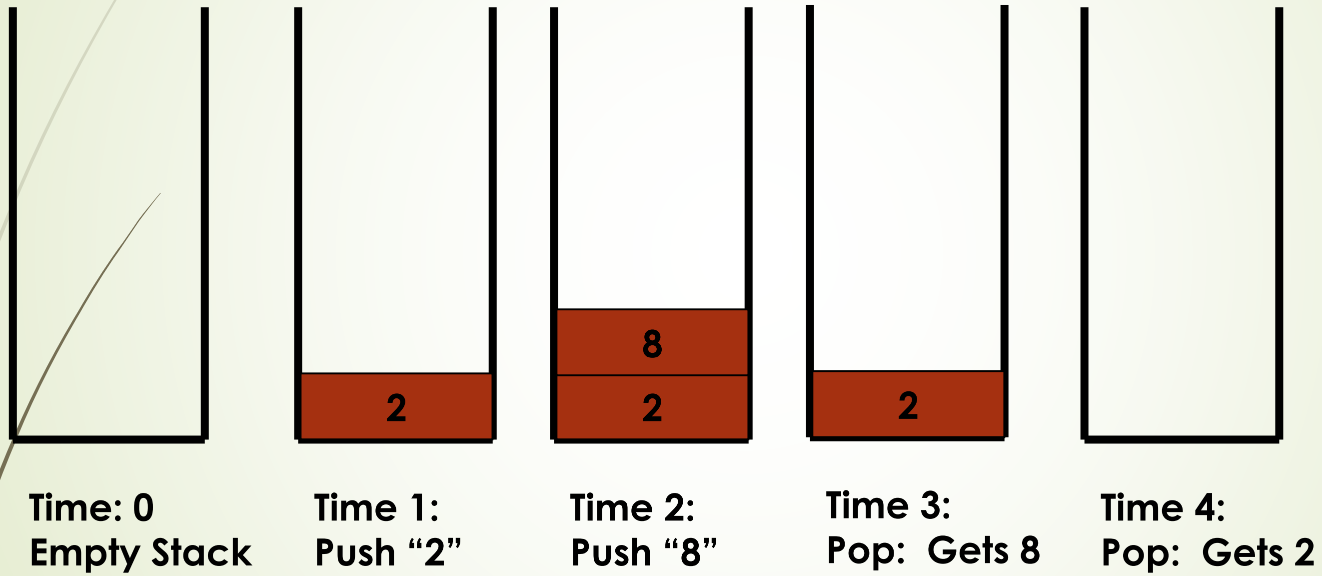
- In computer science, some problems are more easily solved by using recursive functions.
- If you go on to take a computer science algorithms course, you will see lots of examples of this.
- For example:
  - Traversing through a directory or file system.
  - Traversing through a tree of search results.
- For today, we will focus on the basic structure of using recursive methods.

Visualizing Recursion

- To understand how recursion works, it helps to visualize what's going on.
- To help visualize, we will use a common concept called the *Stack*.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - Push: you can push something onto the stack.
  - Pop: you can pop something off the top of the stack.
- Let's see an example stack in action.

Stacks

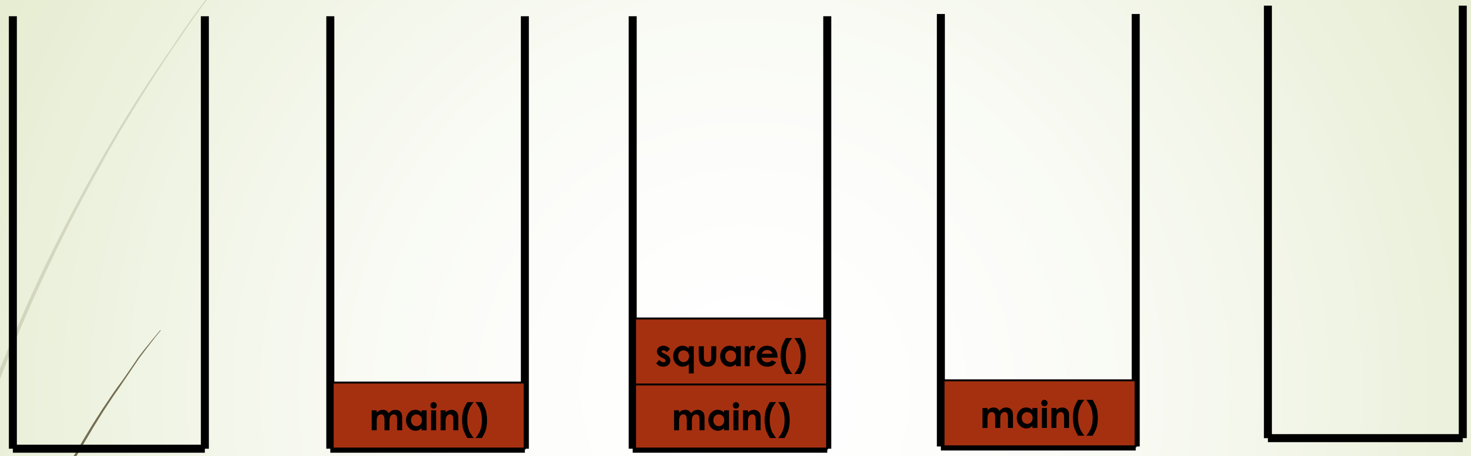
The diagram below shows a stack over time.
We perform two pushes and one pop.



Stacks and Methods

- When you run a program, the computer creates a stack for you.
- Each time you invoke a method, the method is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple C++ program.

Stacks and Methods



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: square()

Time 3:
Pop: square()
returns a value.
method exits.

Time 4:
Pop: main()
returns a value.
method exits.

World's Simplest Recursion Program

World's Simplest Recursion Program

```
void main ( )
  {
   count(0);

  }

  void count (int index)
  {
   cout<<(index);
   if (index < 2)
   count(index+1);
  }
```

This program simply counts from 0-2:
012

This is where the recursion occurs. You can see that the count() function calls itself.

What will be the output

```
void main ( )
  {
   count(0);
  }

  void count (int index)
  {
   cout<<(index);
   if (index < 2)
       count(index+1);
   cout <<(index);
  }
```

012210

Stacks and Recursion

- Each time a method is called, you *push* the method on the stack.
- Each time the method returns or exits, you *pop* the method off the stack.
- If a method calls itself recursively, you just push another copy of the method onto the stack.
- We therefore have a simple way to visualize how recursion really works.

Back to the Simple Recursion Program

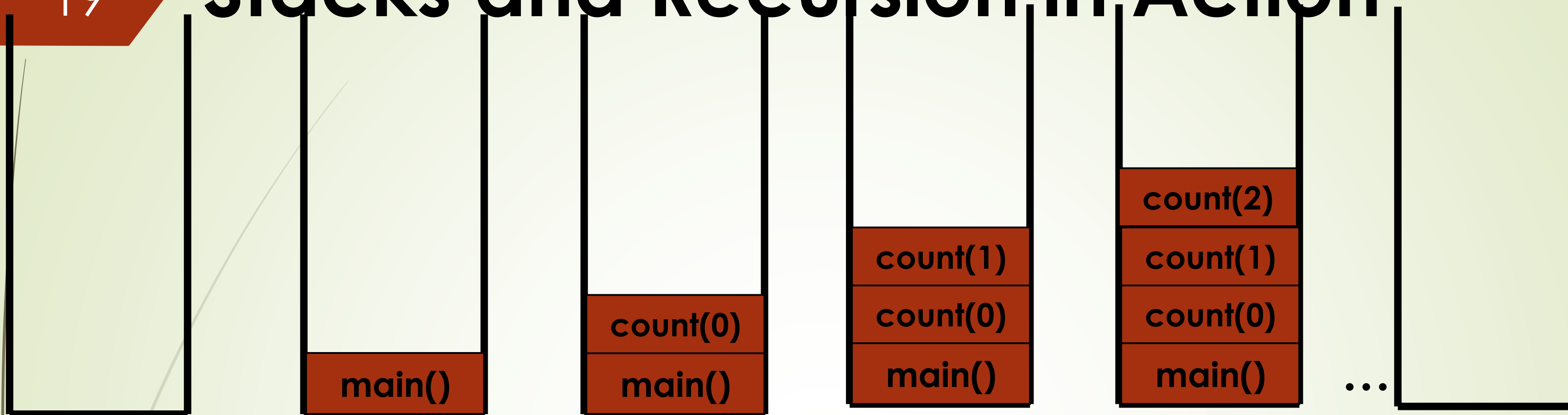
- .

```
void main (void)
   {
    count(0);

   }

   void count (int index)
   {
    cout<<index;
    if (index < 2)
    count(index+1);
   }
```

Stacks and Recursion in Action



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: count(0)

Time 3:
Push: count(1)

Time 4:
Push: count(2)

Times 5-8:
Pop everything

```
Inside count(0):
print (index);        → 0
if (index < 2)
    count(index+1);
```

```
Inside count(1):
print (index);        → 1
if (index < 2)
    count(index+1);
```

```
Inside count(2):
print (index);        → 2
if (index < 2)
    count(index+1);
```

This condition now fails!

Hence, recursion stops, and we proceed to pop all functions off the

Recursion Example #2

Recursion Example #2

```
void main (void)
  {
   upAndDown(1);

  }

  void upAndDown (int n)
  {
   cout<<“level:”;
   cout<<  n;
   if (n < 4)
   upAndDown (n+1);
   cout<<"\nLEVEL: " ;
   cout<< n;
  }
```

Recursion occurs here.

Determining the Output

- Suppose you were given this problem on the final exam, and your task is to “determine the output.”
- How do you figure out the output?
- Answer: Use Stacks to Help Visualize

Stack Short-Hand

- Rather than draw each stack like we did last time, you can try using a short-hand notation.

time	stack	output
time 0:	empty stack	
time 1:	f(1)	Level: 1
time 2:	f(1), f(2)	Level: 2
time 3:	f(1), f(2), f(3)	Level: 3
time 4:	f(1), f(2), f(3), f(4)	Level: 4
time 5:	f(1), f(2), f(3)	LEVEL: 4
time 6:	f(1), f(2)	LEVEL: 3
time 7:	f(1)	LEVEL: 2
time 8:	empty	LEVEL: 1

```
void main (void)
  {
   upAndDown(1);

  }

  void upAndDown (int n)
  {
   cout<<“level:”;
   cout<<  n;
   if (n < 4)
   upAndDown (n+1);
   cout<<"\nLEVEL: " ;
   cout<< n;
  }
```

Computing Factorials

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;

- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a pattern.
The pattern is as follows:
You can compute the factorial of any number (n) by taking n and multiplying it by the factorial of (n-1).

For example:
5! = 5 * 4!
(which translates to 5! = 5 * 24 = 120)

Iterative Approach

```
void main ()
  {
   int answer, n;
   cout<<"Enter a number= "
   cin>>n ;
   answer = findFactorial (n);
  }

int findFactorial (int n)
  {
   int i, factorial = n;
   for (i = n - 1; i >= 1; i--)
   factorial = factorial * i;
   return factorial;
  }
```

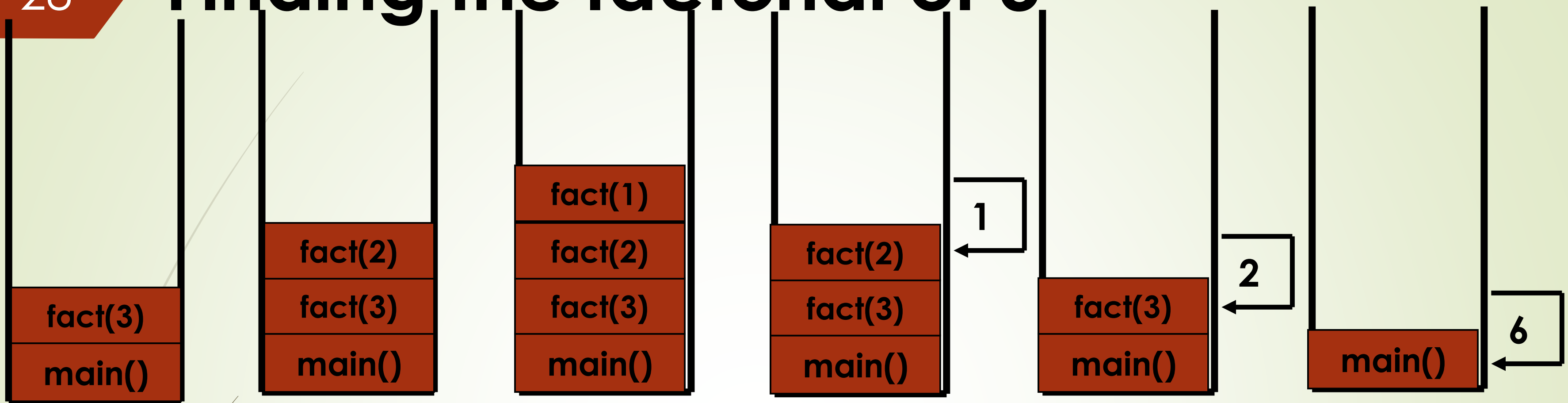
This is an iterative solution to finding a factorial.
It's iterative because we have a simple for loop.
Note that the for loop goes from n-1 to 1.

Recursive Solution

```
int findFactorial (int number)
   {
    if (  (number == 1)   | | (number == 0)  )
         return 1;
    else
         return (number * findFactorial (number-1));
   }
```

Base Case.

Finding the factorial of 3



Time 2:
Push: fact(3)

Time 3:
Push: fact(2)

Time 4:
Push: fact(1)

Time 5:
Pop: fact(1) returns 1.

Time 6:
Pop: fact(2) returns 2.

Time 7:
Pop: fact(3) returns 6.

Inside findFactorial(3):
if (number <= 1) return 1;
else return (3 * factorial (2));

Inside findFactorial(2):
if (number <= 1) return 1;
else return (2 * factorial (1));

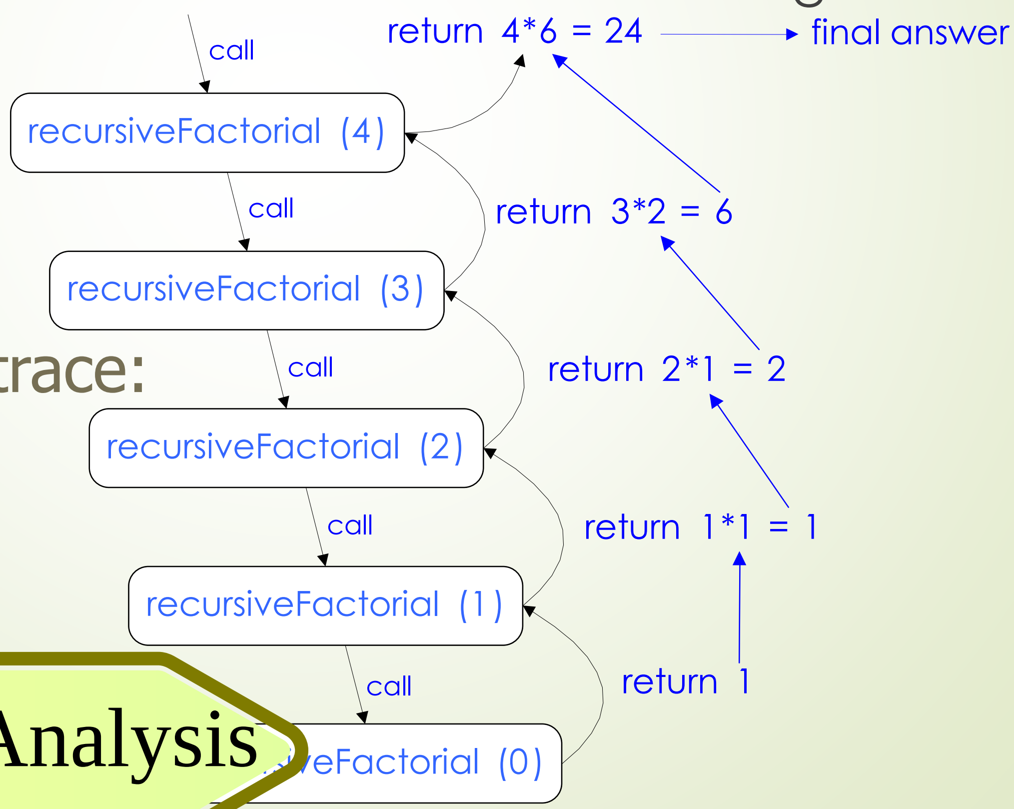
Inside findFactorial(1):
if (number <= 1) return 1;
else return (1 * factorial (0));

Algorithm Analysis

Visualizing Recursion

- Recursion trace
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

Example recursion trace:



Example: The Fibonacci Series

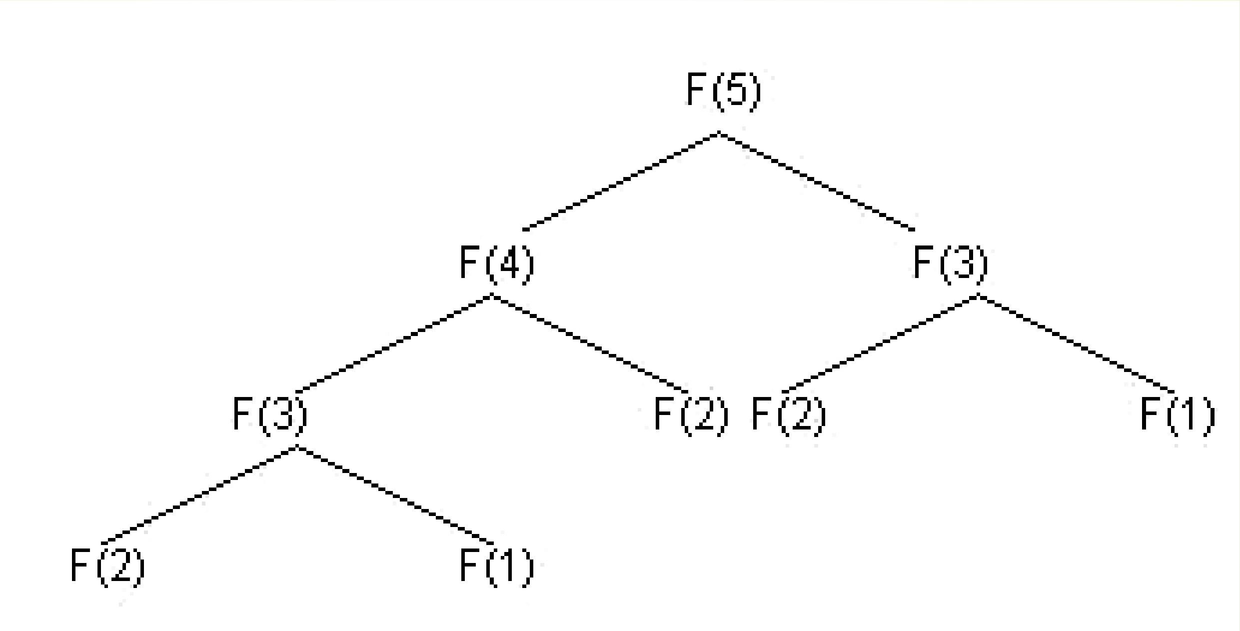
- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)

    - fibonacci(0) and fibonacci(1) are base cases

```
// recursive declaration of method fibonacci
long fibonacci( long n )
{
 if ( n == 0 || n == 1 )
    return n;
 else
 return fibonacci( n - 1 ) + fibonacci( n - 2 );
} // end method fibonacci
```

Recursion Tree showing Fibonacci calls



Recursion vs. Iteration

- Iteration
  - Uses repetition structures (`for`, `while` or `do…while`)
  - Repetition through explicitly use of repetition structure
  - Terminates when loop-continuation condition fails
  - Controls repetition by using a counter
- Recursion
  - Uses selection structures (`if`, `if…else` or `switch`)
  - Repetition through repeated method calls
  - Terminates when base case is satisfied
  - Controls repetition by dividing problem into simpler one

Recursion vs. Iteration (cont.)

- Recursion
  - More overhead than iteration
  - More memory intensive than iteration
  - Can also be solved iteratively
  - Often can be implemented with only a few lines of code

Some Uses For Recursion

- Numerical analysis
- Graph theory
- Symbolic manipulation
- Sorting
- List processing
- Game playing
- General heuristic problem-solving
- Tree traversals

Why use it?

PROS

- Clearer logic
- Often more compact code
- Often easier to modify
- Allows for complete analysis of runtime performance

CONS

- Overhead costs

Summary

- Recursion can be used as a very powerful programming tool
- There is a tradeoff between time spent constructing and maintaining a program and the cost in time and memory of execution.

Algorithm Analysis

Algorithm Analysis

Reading Material

- Nell Dale: Chapter # 7
- Schaum's Outlines: Chapter # 6
- D. S. Malik: Chapter # 6
- Tenebaum: Chapter # 7